**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра «Информационные Системы и Технологий (ИСиТ)»**

отчет

**по лабораторной работе №2**

**по дисциплине «Программирование»**

Тема: «Интерфейсы классов, взаимодействие классов, перегрузка операций»

|  |  |  |
| --- | --- | --- |
| Студент гр. 0324 |  | Зимацкий С. Н. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

2022

**Цель работы.**

Разработать и реализовать набор классов:

● Класс базы

● Набор классов ландшафта карты

● Набор классов нейтральных объектов поля

Класс базы должен отвечать за создание юнитов, а также учитывать юнитов, относящихся к текущей базе. Основные требования к классу база:

● База должна размещаться на поле

● Методы для создания юнитов

● Учет юнитов, и реакция на их уничтожение и создание

● База должна обладать характеристиками такими, как здоровье, максимальное количество юнитов, которые могут быть одновременно созданы на базе, и.т.д.

Набор классов ландшафта определяют вид поля. Основные требования к классам ландшафта:

Должно быть создано минимум 3 типа ландшафта

● Все классы ландшафта должны иметь как минимум один интерфейс

● Ландшафт должен влиять на юнитов (например, возможно пройти по клетке с определенным ландшафтом или запрет для атаки определенного типа юнитов)

● На каждой клетке поля должен быть определенный тип ландшафта

Набор классов нейтральных объектов представляют объекты, располагаемые на поле и с которыми могут взаимодействие юнитов. Основные требования к классам нейтральных объектов поля:

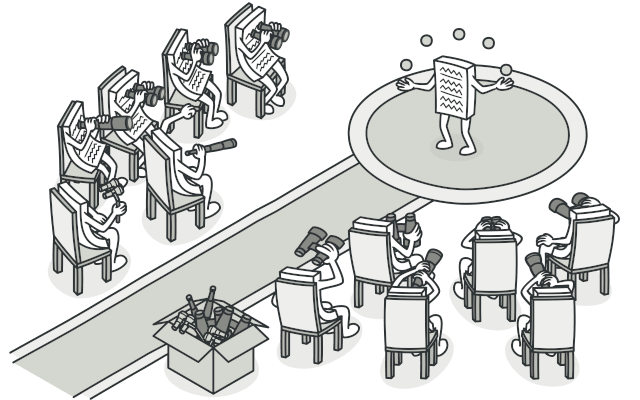
● Создано не менее 4 типов нейтральных объектов

● Взаимодействие юнитов с нейтральными объектами, должно быть реализовано в виде перегрузки операций

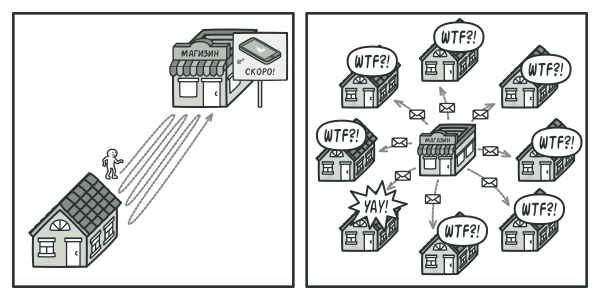
● Классы нейтральных объектов должны иметь как минимум один общий интерфейс

**Основные теоретические положения.**

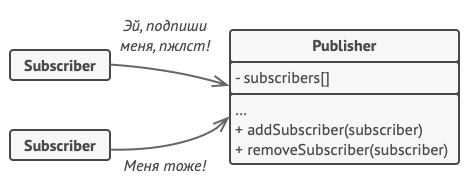
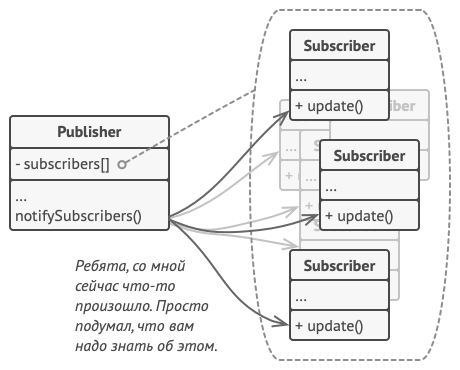
Наблюдатель

* **Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.
* 

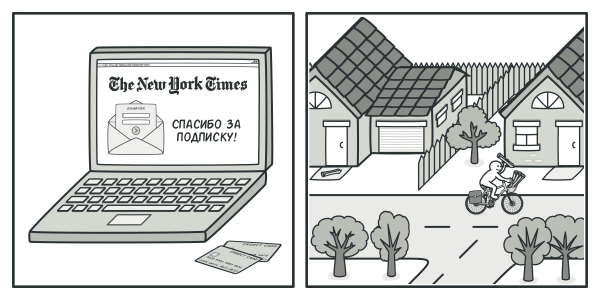
## Проблема

* Представьте, что вы имеете два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю.
* Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.
* 
* другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.
* Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

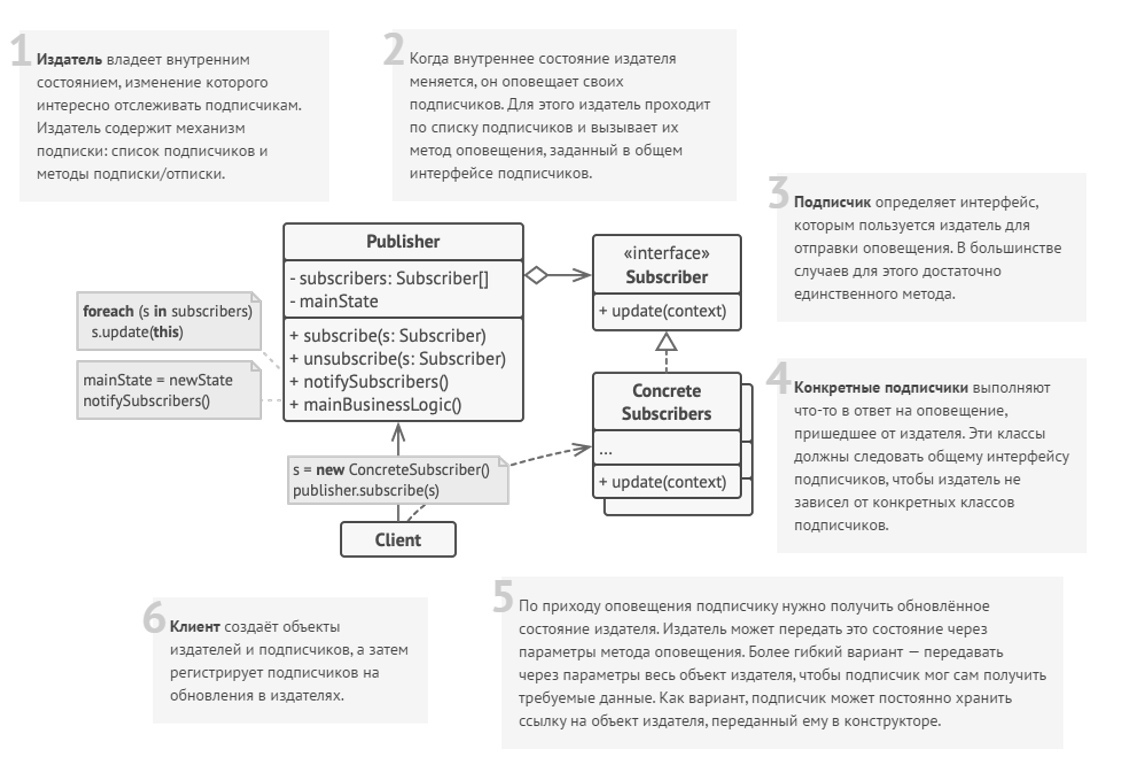
### Решение

* Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.
* Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.
* 
* Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.
* Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.
* 
* Увидев, как складно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.

### Аналогия из жизни

* 
* После того как вы оформили подписку на газету или журнал, вам больше не нужно ездить в супермаркет и проверять, не вышел ли очередной номер. Вместо этого издательство будет присылать новые номера по почте прямо к вам домой сразу после их выхода.
* Издательство ведёт список подписчиков и знает, кому какой журнал высылать. Вы можете в любой момент отказаться от подписки, и журнал перестанет вам приходить.

### Структура

* 

### Применимость

* Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.
* Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.
* Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.
* Когда одни объекты должны наблюдать за другими, но только в определённых случаях.
* Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.

### Шаги реализации

1. Разбейте вашу функциональность на две части: независимое ядро и опциональные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками.
2. Создайте интерфейс подписчиков. Обычно в нём достаточно определить единственный метод оповещения.
3. Создайте интерфейс издателей и опишите в нём операции управления подпиской. Помните, что издатель должен работать только с общим интерфейсом подписчиков.
4. Вам нужно решить, куда поместить код ведения подписки, ведь он обычно бывает одинаков для всех типов издателей. Самый очевидный способ — вынести этот код в промежуточный абстрактный класс, от которого будут наследоваться все издатели.

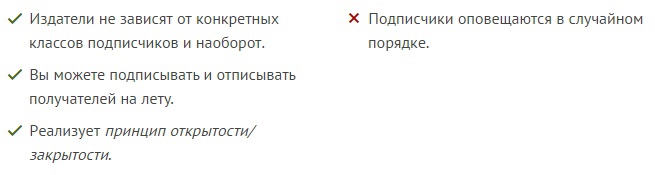
* Но если вы интегрируете паттерн в существующие классы, то создать новый базовый класс может быть затруднительно. В этом случае вы можете поместить логику подписки во вспомогательный объект и делегировать ему работу из издателей.

1. Создайте классы конкретных издателей. Реализуйте их так, чтобы после каждого изменения состояния они отправляли оповещения всем своим подписчикам.
2. Реализуйте метод оповещения в конкретных подписчиках. Не забудьте предусмотреть параметры, через которые издатель мог бы отправлять какие-то данные, связанные с происшедшим событием.

* Возможен и другой вариант, когда подписчик, получив оповещение, сам возьмёт из объекта издателя нужные данные. Но в этом случае вы будете вынуждены привязать класс подписчика к конкретному классу издателя.

1. Клиент должен создавать необходимое количество объектов подписчиков и подписывать их у издателей.

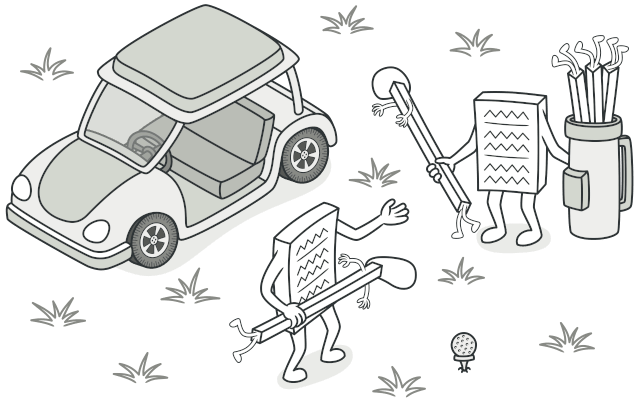
### Преимущества и недостатки

* 

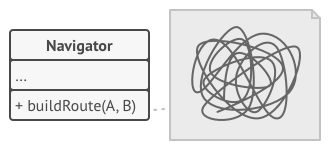
### Отношения с другими паттернами

* Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:
  + Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
  + Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.
  + Посредник убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
  + Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
* Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.
* Цель Посредника — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель Наблюдателя — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.
* Довольно популярна реализация Посредника при помощи Наблюдателя. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.
* Но Посредник имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать Наблюдателя, но всё же останется Посредником.
* Напротив, в случае реализации посредника с помощью Наблюдателя представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети Наблюдателей, не имея центрального объекта-Посредника.

Стратегия

* **Стратегия** — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.
* 

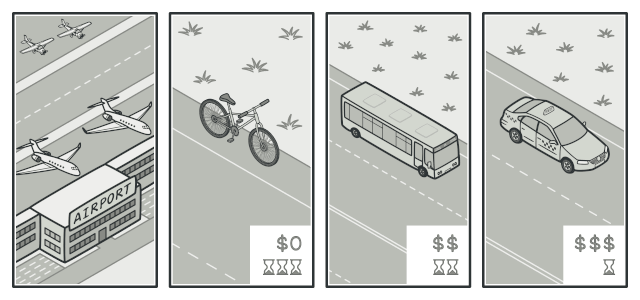
### Проблема

* Вы решили написать приложение-навигатор для путешественников. Оно должно показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.
* Одной из самых востребованных функций являлся поиск и прокладывание маршрутов. Пребывая в неизвестном ему городе, пользователь должен иметь возможность указать начальную точку и пункт назначения, а навигатор — проложит оптимальный путь.
* Первая версия вашего навигатора могла прокладывать маршрут лишь по дорогам, поэтому отлично подходила для путешествий на автомобиле. Но, очевидно, не все ездят в отпуск на машине. Поэтому следующим шагом вы добавили в навигатор прокладывание пеших маршрутов.
* Через некоторое время выяснилось, что некоторые люди предпочитают ездить по городу на общественном транспорте. Поэтому вы добавили и такую опцию прокладывания пути.
* Но и это ещё не всё. В ближайшей перспективе вы хотели бы добавить прокладывание маршрутов по велодорожкам. А в отдалённом будущем — интересные маршруты посещения достопримечательностей.
* 
* Если с популярностью навигатора не было никаких проблем, то техническая часть вызывала вопросы и периодическую головную боль. С каждым новым алгоритмом код основного класса навигатора увеличивался вдвое. В таком большом классе стало довольно трудно ориентироваться.
* Любое изменение алгоритмов поиска, будь то исправление багов или добавление нового алгоритма, затрагивало основной класс. Это повышало риск сделать ошибку, случайно задев остальной работающий код.
* Кроме того, осложнялась командная работа с другими программистами, которых вы наняли после успешного релиза навигатора. Ваши изменения нередко затрагивали один и тот же код, создавая конфликты, которые требовали дополнительного времени на их разрешение.

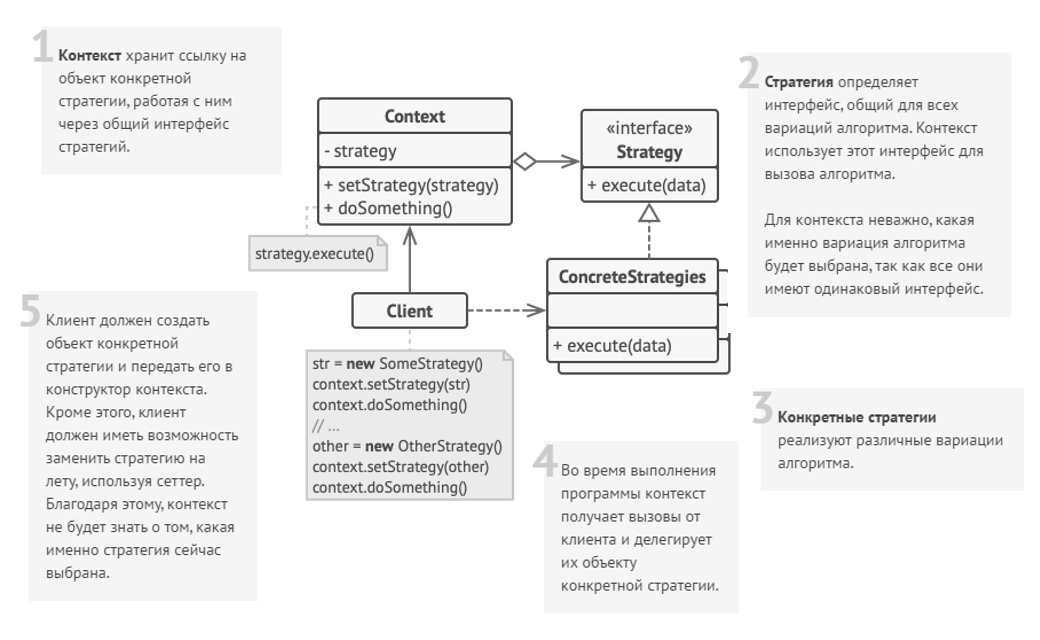
### Решение

* Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.
* Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.
* Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.
* 
* В нашем примере каждый алгоритм поиска пути переедет в свой собственный класс. В этих классах будет определён лишь один метод, принимающий в параметрах координаты начала и конца пути, а возвращающий массив точек маршрута.
* Хотя каждый класс будет прокладывать маршрут по-своему, для навигатора это не будет иметь никакого значения, так как его работа заключается только в отрисовке маршрута. Навигатору достаточно подать в стратегию данные о начале и конце маршрута, чтобы получить массив точек маршрута в оговорённом формате.
* Класс навигатора будет иметь метод для установки стратегии, позволяя изменять стратегию поиска пути на лету. Такой метод пригодится клиентскому коду навигатора, например, переключателям типов маршрутов в пользовательском интерфейсе.

### Аналогия из жизни

* 
* Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией. Вы выбираете конкретную стратегию в зависимости от контекста — наличия денег или времени до отлёта.

### Структура

* 

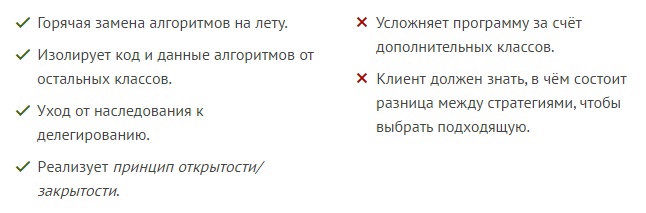
### Применимость

* Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
* Стратегия позволяет варьировать поведение объекта во время выполнения программы, подставляя в него различные объекты-поведения (например, отличающиеся балансом скорости и потребления ресурсов).
* Когда у вас есть множество похожих классов, отличающихся только некоторым поведением.
* Стратегия позволяет вынести отличающееся поведение в отдельную иерархию классов, а затем свести первоначальные классы к одному, сделав поведение этого класса настраиваемым.
* Когда вы не хотите обнажать детали реализации алгоритмов для других классов.
* Стратегия позволяет изолировать код, данные и зависимости алгоритмов от других объектов, скрыв эти детали внутри классов-стратегий.
* Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет собой вариацию алгоритма.
* Стратегия помещает каждую лапу такого оператора в отдельный класс-стратегию. Затем контекст получает определённый объект-стратегию от клиента и делегирует ему работу. Если вдруг понадобится сменить алгоритм, в контекст можно подать другую стратегию.

### Шаги реализации

1. Определите алгоритм, который подвержен частым изменениям. Также подойдёт алгоритм, имеющий несколько вариаций, которые выбираются во время выполнения программы.
2. Создайте интерфейс стратегий, описывающий этот алгоритм. Он должен быть общим для всех вариантов алгоритма.
3. Поместите вариации алгоритма в собственные классы, которые реализуют этот интерфейс.
4. В классе контекста создайте поле для хранения ссылки на текущий объект-стратегию, а также метод для её изменения. Убедитесь в том, что контекст работает с этим объектом только через общий интерфейс стратегий.
5. Клиенты контекста должны подавать в него соответствующий объект-стратегию, когда хотят, чтобы контекст вёл себя определённым образом.

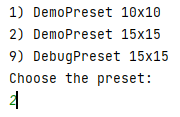
### Преимущества и недостатки

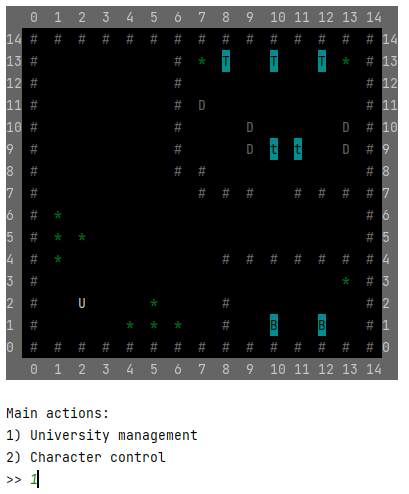
* 

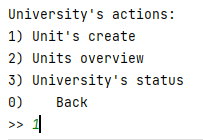
## Отношения с другими паттернами

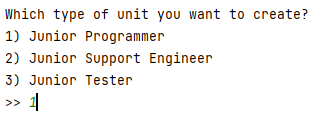
* Мост, Стратегия и Состояние (а также слегка и Адаптер) имеют схожие структуры классов — все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны — это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.
* Команда и Стратегия похожи по духу, но отличаются масштабом и применением:
  + Команду используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.
  + С другой стороны, Стратегия описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.
* Стратегия меняет поведение объекта «изнутри», а Декоратор изменяет его «снаружи».
* Шаблонный метод использует наследование, чтобы расширять части алгоритма. Стратегия использует делегирование, чтобы изменять выполняемые алгоритмы на лету. Шаблонный метод работает на уровне классов. Стратегия позволяет менять логику отдельных объектов.
* Состояние можно рассматривать как надстройку над Стратегией. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в Стратегии эти объекты не знают друг о друге и никак не связаны. В Состоянии сами конкретные состояния могут переключать контекст.

**Экспериментальные результаты.**

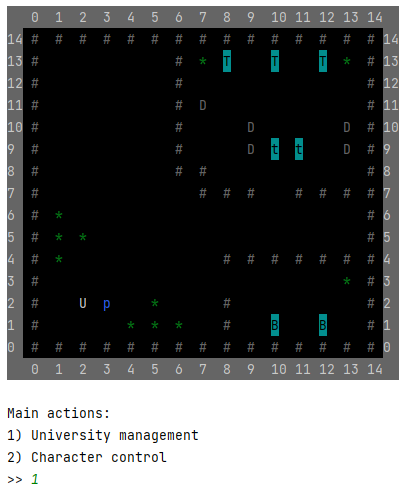


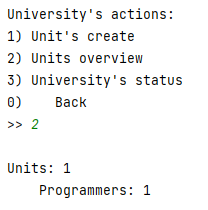


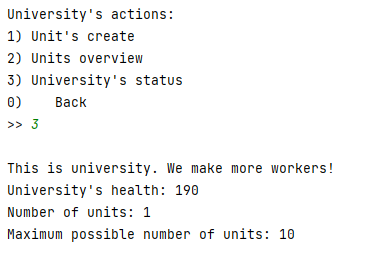


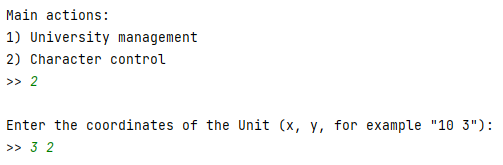


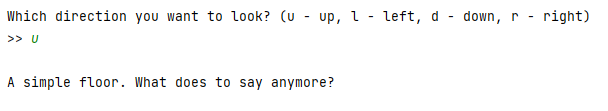


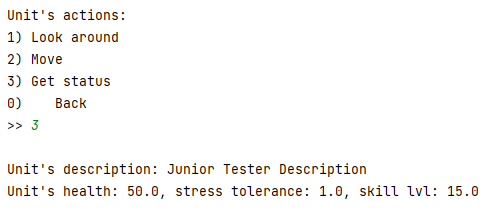


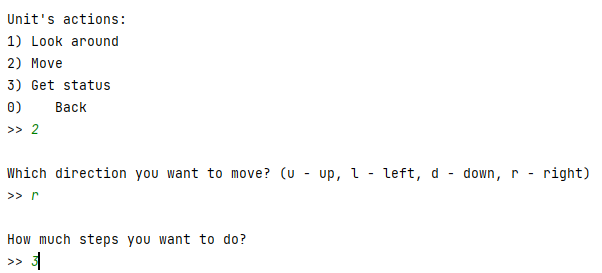


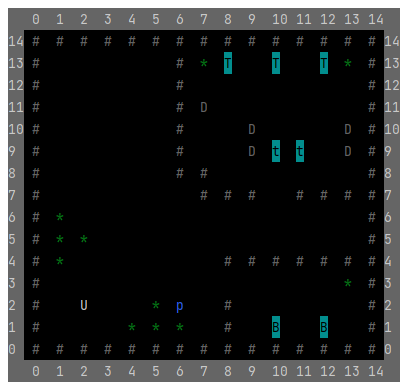










  
Рисунок 5 – Результат работы программы (полный код программы представлен в приложении А)

Приложение а  
полный код программы

<https://github.com/Celeron404/leti2021-2/tree/master/src/main/java/com/leti2021_2>